

# MODELING AND VERIFICATION OF A PIPELINED CPU

Lubomir Ivanov

Department of Computer Science, Iona College, 715 North Avenue, New Rochelle, NY 10801

[livanov@iona.edu](mailto:livanov@iona.edu)

## ABSTRACT

In this paper we present a formal model of a pipelined version of the DLX processor, and verify the correct operation of the pipeline using a formal verification approach based series-parallel posets. We illustrate how the method can be used to detect pipeline hazard and other problems. The full verification was carried out automatically with the help of a verification tool, based on algorithms with low time- and space complexity.

## 1. INTRODUCTION

Formal verification is an area of research, which has recently seen a rapid growth, and has been accepted by the industry as a standard phase in the design and development of complex hardware and software systems. Several powerful verification methods, such as Symbolic Model Checking [1] and Automata Verification [2], have led to the development of software tools for automatic verification of design correctness. These tools are used extensively by the computer and chip-manufacturing industry. Other verification methods trade some degree of expressiveness for a significantly improved running time and storage requirements. Many of these methods have been based on the notion of partial orders [3-7]. The main appeal of using partial orders is in avoiding the study of all possible interleavings of events in a system. In [8] we introduced a new partial order verification method, referred to as *series-parallel poset verification*. Since then the core theory has been significantly extended and implemented into a formal verification software tool, which has been used to carry out verification of various complex systems and protocols, including the PCI bus protocol, the MESI cache coherence protocol, and the T414 Transputer microcode [8-15].

In this paper we consider the issues of modeling and verification of a pipelined version of the DLX processor. We begin with a brief description of DLX and its integer pipeline. We then present a formal model of the pipeline using series-parallel poset expressions, and discuss the verification of pipeline properties. We also present a brief overview of series-parallel poset verification to help the reader follow the presented material. We conclude with a few brief notes on the strengths and weaknesses of our approach in the context of similar verification methods, and an outline of current and future research directions.

## 2. THE DLX PROCESSOR

The DLX processor was introduced by Hennessy and Patterson in [16]. It incorporates many features of several popular commercial microprocessors such as Intel i860, SPARCstation-1, etc. Architecturally, it has thirty-two 32-bit general purpose registers (R0 hardwired to 0), thirty-two floating-point registers, which can be used for single-precision or (in pairs) for double precision floating point computation, and a set of special purpose registers for accessing status information. Memory is accessed through loads and stores which can transfer a byte, a halfword, or a word. The address is 32 bits wide. The instruction set includes four types of instructions:

- *Immediate (I-type)*, (e.g. LW R1,10(R4), or ADDI R1, R2, #4)
- *Register (R-type)*, (e.g. ADDR R1, R2, R3)
- *Jump (J-type)*, (e.g. JR R5, or J NEXT)
- *Floating Point Operations*, (e.g. ADDD F0, F1, F2)

For further architectural details, refer to [16]. Hennessy and Patterson consider several different, progressively more complex and thorough implementations of an integer pipeline for DLX. To illustrate the verification and error checking capabilities of our methodology, we elected to model one of the simpler versions of the DLX integer pipeline, in which the issues of pipeline hazards have not been thoroughly addressed. The datapath of the selected DLX version is given in Fig.1, and the control sequence, which governs its operation is given below in register transfer notation:

*Instruction Fetch (IF):*

IF/ID.IR  $\leftarrow$  Mem[PC]  
if (Cond) PC & IF/ID.NPC  $\leftarrow$  PC + EX/MEM.ALUout  
else PC & IF/ID.NPC  $\leftarrow$  PC + 1

*Instruction Decode (ID):*

ID/EX.A  $\leftarrow$  Regs[IF/ID.IR<sub>6..10</sub>];  
ID/EX.B  $\leftarrow$  Regs[IF/ID.IR<sub>11..15</sub>];  
ID/EX.Imm  $\leftarrow$  ( IF/ID.IR<sub>16</sub> )<sup>16</sup>## IF/ID.IR<sub>16..31</sub>;  
ID/EX.NPC  $\leftarrow$  IF/ID.NPC ; ID/EX.IR  $\leftarrow$  IF/ID.IR ;  
Control  $\leftarrow$  IF/ID.IR<sub>0..5</sub><sup>1</sup>

*Execute (EX):*

- Mem. Ref.: EX/MEM.ALUout  $\leftarrow$  ID/EX.A + ID/EX.Imm
- R-R Instr.: EX/MEM.ALUout  $\leftarrow$  ID/EX.A op ID/EX.B
- R-I Instr.: EX/MEM.ALUout  $\leftarrow$  ID/EX.A op ID/EX.Imm
- Branch: EX/MEM.ALUout  $\leftarrow$  ID/EX.NPC + ID/EX.B ;  
EX/MEM.Cond  $\leftarrow$  (A op 0)<sup>2</sup> ;  
EX/MEM.B  $\leftarrow$  ID/EX.B ; EX/MEM.IR  $\leftarrow$  ID/EX.IR

*Memory Access (MEM):*

- Load: MEM/WB.LMD  $\leftarrow$  Mem[EX/MEM.ALUout]
- Store: Mem[EX/MEM.ALUout]  $\leftarrow$  EX/MEM.B  
MEM/WB.IR  $\leftarrow$  EX/MEM.IR ;  
MEM/WB.ALUout  $\leftarrow$  EX/MEM.ALUout

*Write-Back (WB):*

- R-R.Instr.: Regs[MEM/WB.IR<sub>6..10</sub>]  $\leftarrow$  MEM/WB.ALUout
- R-I Instr.: Regs[MEM/WB.IR<sub>11..15</sub>]  $\leftarrow$  MEM/WB.ALUout
- Load Instr.: Regs[MEM/WB.IR<sub>6..10</sub>]  $\leftarrow$  MEM/WB.LMD

In the above, IF/ID, ID/EX, EX/MEM, and MEM/WB are pipeline registers used to synchronize the operation of the pipeline and to store intermediate values needed in the next phase of the computations. The pipeline registers contain a number of fields corresponding to the usual special purpose registers of a CPU - Instruction Register (IR), Program Counter (PC), ALU Output Register (ALUout), Immediate Register (Imm) for storing immediate values, etc.

<sup>1</sup> All of the microinstructions in the ID phase are executed concurrently.

<sup>2</sup> The branch microinstructions of EX phase are executed concurrently.

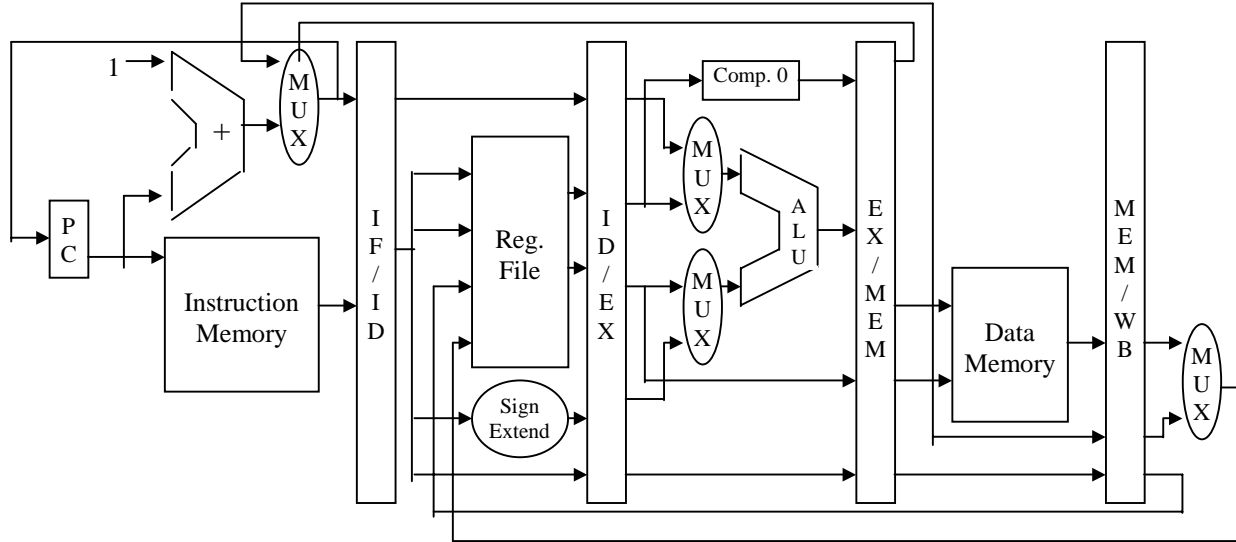


Fig.1 The datapath of a pipelined DLX processor

Data retrieved from the source registers in the register file is loaded during the ID phase in the register file ports A and B, which are also fields in the pipeline registers. LMD is the load memory data register, used to store incoming data from memory. It is a part of the MEM/WB pipeline register. Finally, the Cond field of EX/MEM is a one bit field indicating the outcome of comparing the value in A with 0.

### 3. MODELING THE DLX PIPELINE

A close examination of the pipeline control sequence above reveals that while certain microinstructions must be executed in strict order, others are performed concurrently. Our series-parallel poset methodology is aimed precisely at modeling sequencing and concurrency of events. Hence, it provides a convenient way to model the pipelined DLX processor presented above, and verify the correct operation of its pipeline. In the context of our model, the activation of every control signal is defined to be an *event*. To specify the correct ordering or the independence of events, we use the operations concatenation,  $\bullet$ , (for sequencing), shuffle,  $\otimes$ , (for independence), Kleene star,  $*$ , (for iteration/repetition), and union,  $+$ , (for choice)<sup>3</sup>. The formal model of the DLX control is presented below:

$$\begin{aligned}
 B &= (\text{clk}_H \bullet (\text{IF}_R \otimes \text{ID}_R \otimes \text{EX}_R \otimes \text{MEM}_R \otimes \text{WB}_R) \bullet \text{clk}_L \bullet (\text{IF}_W \otimes \\
 &\quad \text{ID}_W \otimes \text{EX}_W \otimes \text{MEM}_W \otimes \text{WB}_W))^* \quad , \text{ where:} \\
 \text{IF}_R &= \text{PC\_rd} \bullet \text{imr} \otimes \text{EX/MEM.Cond\_rd} \otimes \text{PC\_inc} \\
 \text{ID}_R &= \text{IF/ID.IR\_rd} \bullet \text{Regs\_rd} \otimes \text{IF/ID.NPC\_rd} \\
 \text{EX}_R &= (\text{ID/EX.A\_rd} \otimes \text{ID/EX.B\_rd} \otimes \text{ID/EX.Imm\_rd} \otimes \\
 &\quad \text{ID/EX.NPC\_rd}) \bullet (\text{mux1} \otimes \text{mux2}) \bullet \text{ALU\_op} \otimes \text{comp\_op} \otimes \\
 &\quad \text{ID/EX.IR\_rd} \\
 \text{MEM}_R &= \text{EX/MEM.ALUout\_rd} \bullet (\text{dmr} + \text{EX/MEM.B\_rd}) \otimes \\
 &\quad \text{EX/MEM.IR\_rd} \\
 \text{WB}_R &= \text{MEM/WB.ALUout\_rd} + \text{MEM/WB.LMD\_rd} \\
 \text{IF}_W &= \text{IF/ID.IR\_wr} \bullet (\text{PC\_wr} \otimes \text{IF/ID.NPC\_wr}) \\
 \text{ID}_W &= \text{ID/EX.A\_wr} \otimes \text{ID/EX\_wr.B} \otimes \text{ID/EX.Imm\_wr} \otimes \\
 &\quad \text{ID/EX.NPC\_wr} \otimes \text{ID/EX.IR\_wr} \otimes \text{Control\_decode} \\
 \text{EX}_W &= \text{EX/MEM.ALUout\_wr} \otimes \text{EX/MEM.Cond\_wr} \otimes \\
 &\quad \text{EX/MEM.B\_wr} \otimes \text{EX/MEM.IR\_wr}
 \end{aligned}$$

$$\begin{aligned}
 \text{MEM}_W &= (\text{MEM/WB.LMD\_wr} + \text{dmw}) \otimes \text{MEM/WB.IR\_wr} \otimes \\
 &\quad \text{MEM/WB.ALUout\_wr}
 \end{aligned}$$

$$\text{WB}_W = \text{MEM/WB.IR\_rd} \bullet \text{Regs\_wr}$$

The interpretation of the above expressions is that all five stages of the instruction cycle are carried out concurrently - one stage for each of the five instructions in the pipeline. In each stage, different actions are taken during the first and the second part of the clock ( $\text{clk}_H$  &  $\text{clk}_L$ ). The description of the particular events is given in the table below:

Event Label	Event Description
$\text{clk}_H$	Clock goes high (first half of the cycle)
PC_rd	The current value of PC is read
imr	Instr. Memory Read Control is activated
EX/MEM.Cond_rd	Cond. field of EX/MEM is read
PC_inc	1 is added to the current value of PC
x.IR_rd	The value of IR is read from pipeline reg. x
Regs_rd	The Register File is read from
x.NPC_rd	The current value of NPC in x is read
x.A_rd	The value of Port A is read from x
x.B_rd	The value of Port B is read from x
x.Imm_rd	The value of the Immediate register is read
x.NPC_rd	The value of NPC is read from x
mux1	The MUX at ALU input 1 selects an input
mux2	The MUX at ALU input 2 selects an input
ALU_op	ALU carries out appropriate operation
comp_op	The comparator compares A with 0
x.ALUout_rd	The ALUout field of pipeline reg. x is read
dmr	Data Memory Read Control is activated
MEM/WB.LMD_rd	Read the value of LMD from MEM/WB
x.IR_wr	Write a new value to the IR field of x
PC_wr	Write a new value to the PC
x.NPC_wr	Update the NPC field of pipeline reg. x
x.A_wr	Load (write) a new value into Port A
x.B_wr	Load (write) a new value into Port A
x.Imm_wr	Write a new value into the Immediate reg.
Control_decode	Control Unit decodes opcode in $\text{IR}_{0,5}$
x.ALUout_wr	Write the ALU result into ALUout
EX/MEM.Cond_wr	Update Cond with the result of the compare
MEM/WB.LMD_wr	Write the data from memory into LMD
dmw	Data Memory Write Control is activated
Regs_wr	The Register File is written to.

<sup>3</sup> For reference consult the Overview section of this paper or ref. [13].

#### 4. VERIFYING PIPELINE PROPERTIES

Verifying the correct operation of the DLX pipeline requires the specification of a set of properties, which test all aspects of system operation. Some of these properties (such as Property  $P_1$  below) must always be satisfied, regardless of whether or not pipelining is employed in the design. Other properties are pipeline specific. They test the pipeline for structural, data, and control hazards. Once the properties are identified and converted to series-parallel poset expression format, they can be verified using the developed software package based on the theory defined in [13] and briefly outlined in the "Overview" section below. For lack of space, we limit ourselves to demonstrating the verification of only one property:

*Property  $P_1$ :* "Memory addresses remain fixed for the entire duration of a memory access."

Memory is accessed during an instruction fetch or during a data read (load) or write (store). During an instruction fetch, the address is held stable by the PC register. Its value must not be changed until the fetched instruction is written to the IR field of the IF/ID pipeline register. In other words, IF/ID.IR\_wr must precede PC\_wr. During a data read or write, the effective address for the memory access is computed during the EX phase by the ALU, and maintained fixed maintained fixed by the ALUout field of the EX/MEM pipeline register. This address should not change until either the LMD register records the data coming from memory, or the data memory write control signal is activated and the data is written to memory, i.e. the events MEM/WB.LMD\_wr and dmw must precede the event EX/MEM.ALUout\_wr. The property must be consistently true in each iteration. Thus, property  $P_1$  can be expressed as the following series-parallel poset expression:

$$P_1 = ((\text{IF/ID.IR\_wr} \bullet \text{PC\_wr}) \otimes ((\text{MEM/WB.LMD\_wr} + \text{dmw}) \bullet \text{EX/MEM.ALUout\_wr}))^*$$

The reduced behavior<sup>4</sup> with respect to the events in  $P_1$  is:

$$B = ((\text{IF/ID.IR\_wr} \bullet \text{PC\_wr}) \otimes (\text{EX/MEM.ALUout\_wr} \otimes (\text{MEM/WB.LMD\_wr} + \text{dmw})))^*$$

We would like to confirm that property  $P_1$  is always satisfied. Entering the behavior,  $B$ , and the property,  $P_1$ , in our verification environment (Fig.2), and selecting the "Always Satisfied" option returns FALSE, i.e. the property is not always satisfied. Choosing option "Sometimes Satisfied" however returns TRUE.



Fig.2 SPPV Verification Environment

A careful analysis reveals that the problem lies in the data load/store part of the behavior of the DLX pipeline. The behavior specifies that updating the value of EX/MEM.ALUout is done independently of accessing the data memory. That means that it is sometimes possible for the value of EX/MEM.ALUout to be updated before the data is written into MEM/WB.LMD or into

memory. The property however requires strict sequencing of the three events such that EX/MEM.ALUout is changed only after the memory access is complete. Thus the property is not always satisfied and the verification software returns FALSE. Verifying property  $P_1$  reveals a structural hazard in the DLX pipeline. Notice that in the non-pipelined CPU this is not a problem since the memory access phase of the instruction cycle follows the execute phase, which is the only time when EX/MEM.ALUout is modified. To verify this claim, we can rewrite the behavior as follows:

$$B = (\text{clk}_H \bullet \text{IF}_R \bullet \text{ID}_R \bullet \text{EX}_R \bullet \text{MEM}_R \bullet \text{WB}_R \bullet \text{clk}_L \bullet \text{IF}_W \bullet \text{ID}_W \bullet \text{EX}_W \bullet \text{MEM}_W \bullet \text{WB}_W)^*$$

In this case, the stages are executed in order, without overlap. Entering this behavior and property  $P_1$  in our verification environment confirms that in the non-pipelined DLX,  $P_1$  is always satisfied.

#### 5. OVERVIEW OF SERIES-PARALLEL POSETS

A *partially ordered set (poset)* is a set with a reflexive, antisymmetric, and transitive relation defined on the set elements. A  $\Sigma^*$ -labeled poset  $P=(P, \leq, l)$  consists of a poset  $(P, \leq)$ , and an assignment of a nonempty word (a label)  $l(v) \in \Sigma^*$  to each vertex  $v$  in  $P$ . Given posets  $P$  and  $Q$  ( $P \cap Q = \emptyset$ ), we define two operations:

**Concatenation ( $\bullet$ ):**  $P \bullet Q := (P \cup Q, \leq_{P \bullet Q})$

**Shuffle ( $\otimes$ ):**  $P \otimes Q := (P \cup Q, \leq_{P \otimes Q})$ ,

where:  $v \leq_{P \bullet Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v' \vee (v \in P \wedge v' \in Q)$

$v \leq_{P \otimes Q} v' \Leftrightarrow v \leq_P v' \vee v \leq_Q v'$

*Series-Parallel Poset* over an alphabet  $\Sigma$  is defined inductively:

- The empty poset,  $I$ , is a SPP
- For each  $\sigma \in \Sigma$ , the singleton poset labeled  $\sigma$  is a SPP
- If  $P$  and  $Q$  are SPPs, so are  $P \bullet Q$  and  $P \otimes Q$

Thus, the set of all series parallel posets formed from  $I$  and the singletons and closed under concatenation ( $\bullet$ ) and shuffle ( $\otimes$ ) forms a bimonoid denoted  $\text{SP}(\Sigma^*)$  [17]. For our purposes, the alphabet,  $\Sigma$ , will consist of all distinct events occurring during a run of the system under consideration. Let each event,  $e_i$ , occurring in a system be represented by a singleton poset,  $e_i$ . The fact that event  $e_i$  precedes event  $e_j$  can be represented by  $e_i \bullet e_j$ . The independence of events  $e_i$  and  $e_j$  is represented by the series-parallel poset  $e_i \otimes e_j$ . This extends naturally to sets of events. Interpreting series-parallel posets as descriptions of the dependence or independence of sets of events allows us to model system *behavior* in terms of event sequences. A *non-iterated system* is one, in which the events are distinct and not repeated. *Non-iterated behavior* can be modeled by a single series-parallel poset over the alphabet of system events [8, 10]. An *iterated system* is one, in which some or all events are repeated one or more times. Thus, an iterated system consists of a number of components, which function in series or independently such that each component is either an iterated- or a non-iterated system. To describe *iterated behavior* we need a new structure - the star shuffle semiring  $\mathcal{S} = (\mathcal{S}, +, \bullet, \otimes, *, \theta, I)$  of series-parallel posets:

- $\mathcal{S}$  - set of finite subsets of  $\text{SP}(\Sigma^*)$  closed under the operations
- If  $K \in \mathcal{S}$  and  $L \in \mathcal{S}$ ,  $K+L = \{P \mid P \in K \vee P \in L\} \in \mathcal{S}$
- If  $K \in \mathcal{S}$  and  $L \in \mathcal{S}$ ,  $K \bullet L = \{P \bullet Q \mid P \in K \wedge Q \in L\} \in \mathcal{S}$
- If  $K \in \mathcal{S}$  and  $L \in \mathcal{S}$ ,  $K \otimes L = \{P \otimes Q \mid P \in K \wedge Q \in L\} \in \mathcal{S}$
- If  $K \in \mathcal{S}$ , then  $K^* = I + K + K \bullet K + K \bullet K \bullet K + \dots$
- $\theta$  is the empty set of posets &  $I$  is the empty poset

<sup>4</sup> See the Overview section or ref.[13].

We define the *behavior*,  $B$ , of an iterated system to be an element of the star-shuffle semiring  $S$ , i.e.  $B \in S$ . Thus, the behavior of an iterated system is a set of series-parallel posets. We can represent the *verification properties* as sets of series-parallel posets over a subset of the alphabet  $\Sigma$ , i.e.  $P \in S$  as well. There are 4 normal forms of behavior and property expressions:

- Concatenation:  $B = B_1 \bullet B_2 \bullet \dots \bullet B_n$  &  $P = P_1 \bullet P_2 \bullet \dots \bullet P_m$
- Shuffle:  $B = B_1 \otimes B_2 \otimes \dots \otimes B_n$  &  $P = P_1 \otimes P_2 \otimes \dots \otimes P_m$
- Plus:  $B = B_1 + B_2 + \dots + B_n$  &  $P = P_1 + P_2 + \dots + P_m$
- Star:  $B = B_1^*$  &  $P = P_1^*$

To simplify the reasoning about sets of events and the complexity of the verification algorithms we introduced the notion of a behavior reduction [10, 13]. It is prompted by the fact that, while the system behavior may involve hundreds of thousands of events, in most cases the verification property involves only a few events. The reduction is carried out by a recursively defined *projection function*  $Pr(B, set(P))$ , which eliminates from the behavior all events not in the property, while preserving the ordering among the remaining events.

The *verification questions* are specified as *predicates*:

- $SS(B, P)$  is a binary predicate, interpreted as “The property  $P$  is sometimes satisfied within the behavior,  $B$ ”. The predicate takes a behavior and a property and verifies that the property can sometimes be traced within the behavior.
- $AS(B, P)$  is a binary predicate, interpreted as “The property  $P$  is always satisfied within the behavior,  $B$ ”. The predicate takes a behavior and a property and verifies that the property can always be traced within the behavior.

The formal definitions of the two predicates are based on a number of theorems, corollaries, and lemmas, which examine the satisfaction of all forms of properties with respect to all forms of behaviors [13]. The two predicates are at the core of the SPPV (Series-Parallel Poset Verification) environment used above.

## 6. CONTRIBUTIONS AND LIMITATIONS

The main strength of our approach is in using partial orders to model the independence of sets of events and the interleaving of their execution in time. The higher the degree of independence among events in the system is, the better the performance of our method. For systems with few or no independent events, the performance of our method is equivalent to that of Automata Verification. Indeed, it can be easily seen that, from a formal language point of view, the series-parallel poset expressions are regular expressions augmented with an additional operation - shuffle,  $\otimes$ . In the best case, however, our method delivers an exponential improvement over the performance of similar approaches based on automata theory. The issues of event sequencing and timing has been studied for a long time by many researchers. Closest to our work is that of V.Pratt [4]. However, the main stress in [4] is on modeling system behavior (communication channels in particular), whereas the aim of our technique is not only modeling, but verification as well.

Though it has major advantages, one important shortcoming of our technique is the inability to model so-called “N”-type dependencies among the events occurring in a system. These are encountered quite often in real systems and significantly limit the general applicability of our algorithm. Currently we are working on extending our verification methodology to deal with this type of event dependencies as well.

## 7. CONCLUSIONS

In this paper we presented some aspects of the verification of a pipelined CPU with the help of a new formal verification method - series-parallel poset verification, and a software verification environment based on that theory. We demonstrated how important CPU properties can be checked and pipeline hazards uncovered. A detailed description of the complete pipelined DLX verification will appear separately in a journal publication.

Current work involves extending the theory of series-parallel poset verification to handle arbitrarily complex systems, including those whose events exhibit “N”-type dependencies. Additional work is being done on augmenting the basic set of operations with an additional operation - iterated shuffle - which will allow the modeling and verification of sets of concurrent processes in a distributed memory MIMD parallel computers.

## 8. REFERENCES

- [1] K. McMillan, “*Symbolic Model Checking*”, Kluwer, 1993
- [2] R.Kurshan, “*Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*”, Princeton Series in Computer Science, Princeton, 1994
- [3] M.Nielsen, G.Plotkin, and G.Winskel, “*Petri nets, event structures, and domains*”, part I, TCS, 13:85-108, 1981
- [4] V.Pratt, “*Modeling Concurrency with Partial Orders*”, International Journal of Parallel Programming, 1986
- [5] P. Godefroid, “*Partial Order Methods for the Verification of Concurrent Systems: an Approach to the State Explosion Problem*”, Doctoral Dissertation, University of Liege, 1995
- [6] R. Nalumasu, G. Gopalakrishnan, “*A New Partial Order Reduction Algorithm for Concurrent System Verification*”, Proc of IFIP, 1996
- [7] D. Peled, “*Combining Partial Order Reductions with On-the-Fly Model Checking*”, Journal of Formal Methods in Systems Design, 8 (1), 1996
- [8] L.Ivanov, R.Nunna, S.Bloom, “*Modeling and Analysis of Non-Iterated Systems: An Approach Based on Series-Parallel Posets*”, Proceedings of ISCAS'99, 1999
- [9] L.Ivanov, R.Nunna, “*Formal Verification with Series-Parallel Posets of Globally-Iterated Locally-Non-Iterated Systems*”, Proceedings .of MWSCAS'99, 1999
- [10] L.Ivanov, R.Nunna, “*Formal Verification: A New Partial Order Approach*”, Proceedings of ASIC/SOC'99, 1999
- [11] L. Ivanov, R. Nunna, “*Modeling and Verification of an Interconnect Bus Protocol*”, Proc. .of MWSCAS'00, 2000
- [12] L.Ivanov, R.Nunna “*Modeling and Verification of Cache Coherence Protocols*”, ISCAS'01, Sydney, Australia, 2001
- [13] L.Ivanov , R.Nunna “*Modeling and Verification of Iterated Systems and Protocols*”, MWSCAS'01, Dayton, OH, 2001
- [14] L.Ivanov “*Formal Verification of a Microprocessor Control*”, MWSCAS'01, Dayton, OH, 2001
- [15] L.Ivanov “*Formal Verification of Microinstruction Sequencing*”, Proceedings of ICCIT'01, Montclair, NJ, 2001
- [16] J.Hennessy, D.Patterson, “*Computer Architecture: A Quantitative Approach*”, 2<sup>nd</sup> edition, Morgan Kaufmann, 1996
- [17] S. Bloom, Z. Esik, “*Free Shuffle Algebras in Language Varieties*”, TCS 163 (1996) 55-98, Elsevier